Minesweeper Report

March 22, 2021

Group Information

Reagan McFarland (rpm141), Alay Shah (acs286), Toshanraju Vysyaraju (tv135)

Work Split

- Representation: Reagan
- Basic Agent: Reagan, Alay, and Toshanraju
- Improved Agent: Reagan, Alay, and Toshanraju
- Questions and Write Up: Reagan, Alay, and Toshanraju

Statements

- Reagan McFarland: On my honor, I have neither received nor given any unauthorized assistance on this assignment.
- Alay Shah: On my honor, I have neither received nor given any unauthorized assistance on this assignment.
- Toshanraju Vysyaraju: On my honor, I have neither received nor given any unauthorized assistance on this assignment.

Representation

How did you represent the board in your program, and how did you represent the information / knowledge that clue cells reveal? How could you represent inferred relationships between cells?

To represent the game board, we created a class called `Board` which holds a 2D array of `BoardTile` instances for the game to work. The `Board` also holds the dimensions of the board, the number of mines, and some extra stuff for us to render the board in `pygames`. The actual information is stored in each instance of `BoardTile` which holds, the i, j position of the tile on the board, the value of the tile (clue), if it has been opened, and if it has been flagged. A tile's clue is the sum of mines that are around it and is calculated when the tile is opened. The `Agent` class can only open a tile or flag a tile, it does not know the value of the tile before it is opened.

Inference

When you collect a new clue, how do you model / process / compute the information you gain from it? In other words, how do you update your current state of knowledge based on that clue? Does your program deduce everything it can from a given clue before continuing? If so, how can you be sure of this, and if not, how could you consider improving it?

In the basic agent, when a new clue is collected, we extract all possible information by looking at just that one clue. In the basic agent, the agent visits a cell to open it, once opened, it will look at it until it can either flag or open all of its neighbors. The only way the agent is able to do this is by opening other cells and getting more information. It is important to note that the basic agent only looks at one clue at a time and is not able to use two clues to come to a conclusion.

Our advanced agent, **Inferential Bomb Squad (IBS)**, is built upon the basic agent. It will only begin its process of using the whole knowledge base to act upon the board if the basic agent is unable to (this effectively removes all solved equations from the knowledge base as nothing new needs to be inferred from them). The advanced agent will build the knowledge base by looking at tiles whose equations are unsatisfied and it will add those equations to the board. The equations are `sympy` equations stored with the value of the equation. The advanced agent then will reduce the equations to their simplest form. In this process, any equations whose free variables are subsets of another equation's free variables will be reduced. For example, if

we have A + B + C + D = 2 and we also have B + C + D = 1, then we can simplify our first equation to A = 1. This will allow us to reduce the redundancy of information in the knowledge base, and may allow us to conclude that certain squares are mines or other squares are free. This is something we would not have been able to do if we just looked at solely one clue.

The advanced agent will take this a step further and will also look at 2 equations that may not have a subset/superset relation. It will look at both equations and try to infer any new information if possible. For example, if we have A + B + C = 2 and B + C + D = 1, alone we cannot infer anything and subset reduction does not work. By subtracting the first equation from the second, we can infer that A - D = 1 and this will tell us that A = 1 and D = 0!

Our knowledge base is updated when necessary, it will be built when the advanced inference is called to avoid inconsistencies with our information. This allows our agent to work quickly when doing basic inference (one clue at a time) and ensures that when it is looking at the entire knowledge base, it is correct.

Our agent does not infer all possible information before continuing. It will only employ more complex methods of inference when necessary. This is to keep it from getting stuck and spending extra resources on information that may be much easier to infer a few moves down the line. If we want to infer all possible information at each step, we would like to employ the advanced inference methods at each turn, not when absolutely necessary. Once these methods infer new information that the agent can act on, we stop. The agent will discover new information and may not need more complicated methods of inference.

If we wanted the agent to infer as much as possible, we would improve the agent by calling the complex inference methods at each turn. In addition, we terminate these methods once we are able to act, thus we do not infer all possible information and may not use all inference methods we have described. Finally, we could employ something like a Constraint Satisfaction Tree or use a Proof by Contradiction at each turn to get all possible information. Our `hyper_advanced agent` uses proof by contradiction, as described in the Efficiency Section of the report.

Decisions

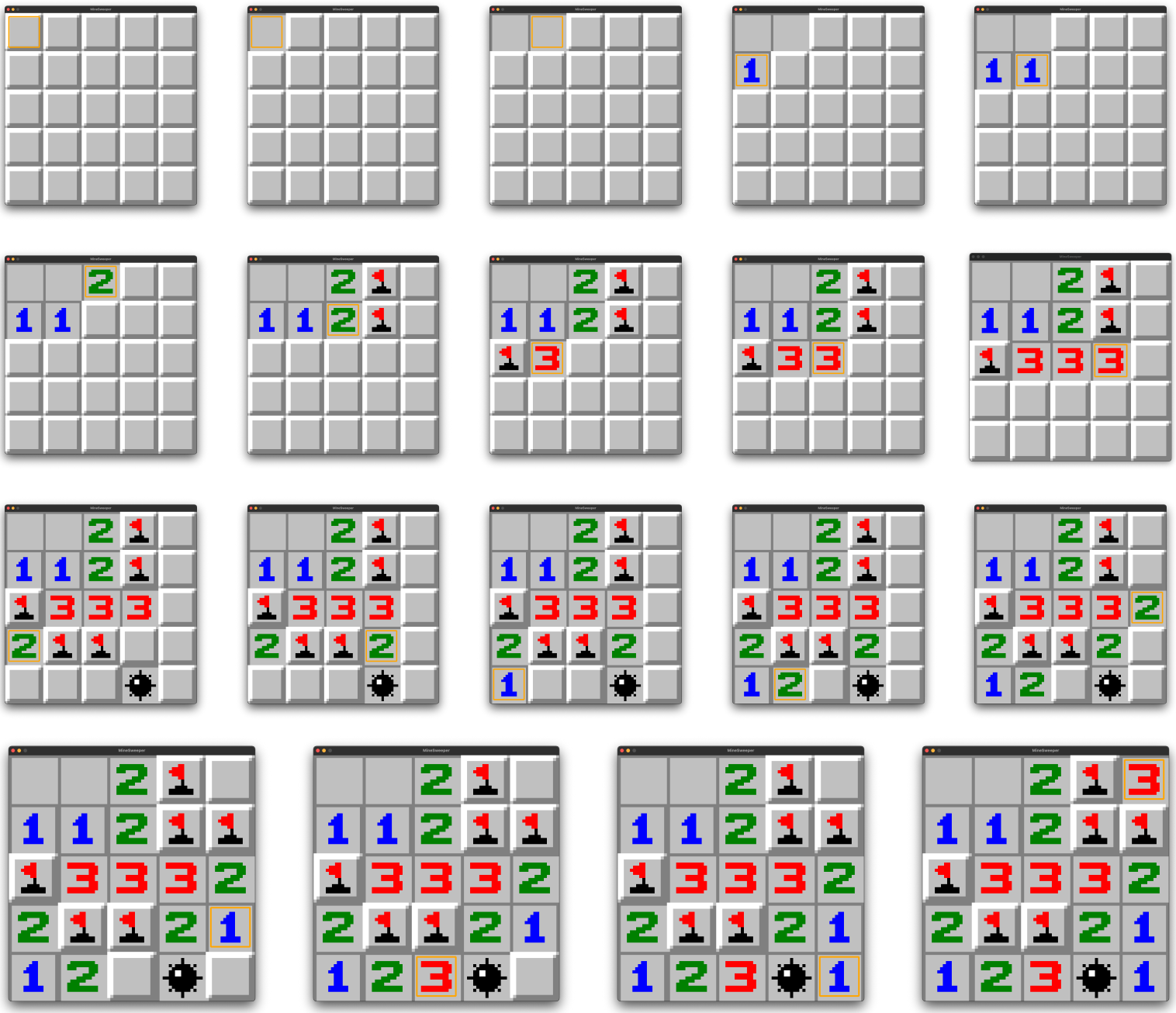
Given a current state of the board, and a state of knowledge about the board, how does your program decide which cell to search next?

The program knows which tile to search next if it definitely knows that this tile is not a mine. All tiles that are known to have a value of 0 will be put in a list of tiles to open next. It will determine this using the inference methods described above. If all inference methods fail and the list of safe, unopened tiles is empty, the algorithm will choose a random unopened tile to search and will again try to determine new information.

Performance

For a reasonably-sized board and a reasonable number of mines, include a play-by-play progression to completion or loss. Are there any points where your program makes a decision that you don't agree with? Are there any points where your program made a decision that surprised you? Why was your program able to make that decision?

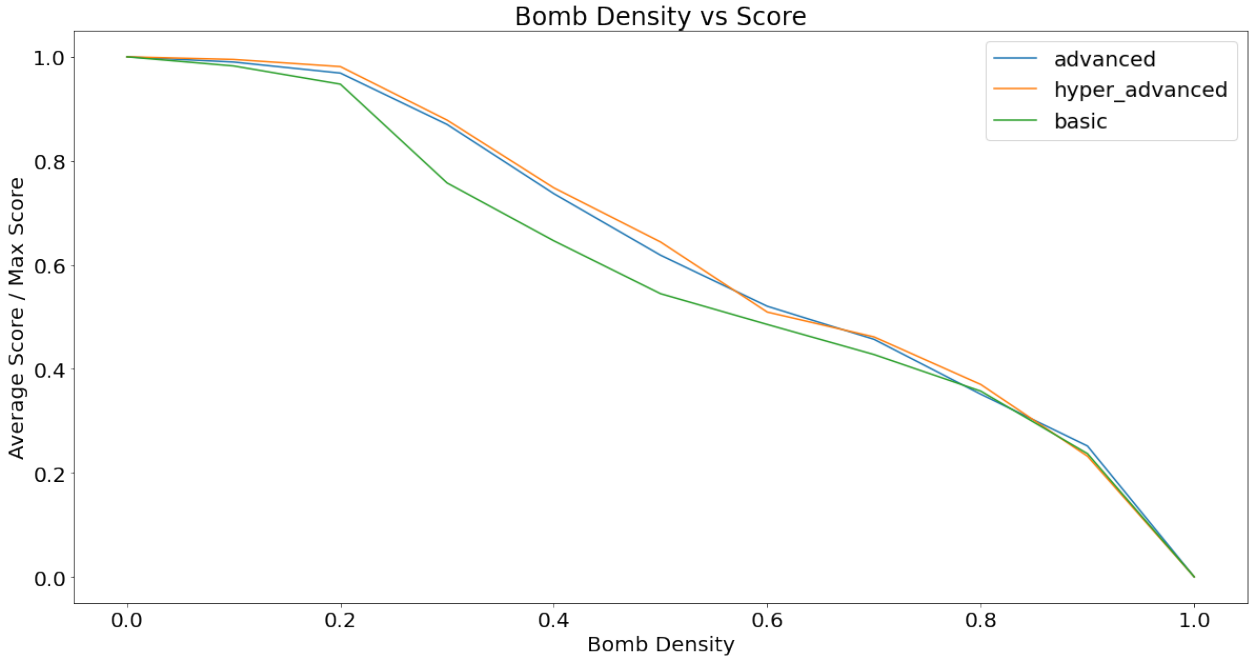
Below is a play-by-play progression to completion of a 5-by-5 board with 7 mines (a .28 mine density). The agent only opens one bomb on screenshot 11. This only occurs because it is forced to open a random tile as it is unable to infer any further information from the board. Our agent did not make any decisions that we do not agree with and it did not make any decisions that surprised us as well. Each step that it took was a step that we would have taken ourselves as well.



For a fixed, reasonable size of board, plot as a function of mine density the average final score (safely identified mines / total mines) for the simple baseline algorithm and your algorithm for comparison. This will require solving multiple random boards at a given density of mines to get good average score results. Does the graph make sense / agree with your intuition? When does minesweeper become 'hard'? When does your algorithm beat the simple algorithm, and when is the simple algorithm better? Why? How frequently is your algorithm able to work out things that the basic agent cannot?

Below is a plot for a 20-by-20 board with increasing bomb density (plotted on the x-axis) for the `basic agent`, our `advanced agent` that uses inference, and our `hyper_advanced agent` that uses inference and proof by contradiction. For each bomb density, we run 10 different boards and take the average number of bombs safely flagged and divide by the total number of bombs on the board (the y-axis). This graph agrees with our intuition since the `advanced agent` and the `hyper_advanced agent` do better than the `basic agent` in general. Minesweeper begins to become 'hard' at around 0.8 bomb density. This is around when all three agents perform about the same. There are way too many bombs so the agents are no longer able to make

helpful inferences about the location of unopened bombs. Our algorithms beat the `basic agent` for bomb densities between 0.0 and 0.8. There is no density where the `basic agent` does better than our agents. Our agents beat the `basic agent` because rather than just using one equation (looking at the information known about just one tile) like the `basic agent` does, the `advanced agent` and the `hyper_advanced agent` look at 2 equations (information known about two tiles) so they are able to infer more information and avoid having to open random tiles as much as the `basic agent`. For the bomb densities between 0.3 and 0.6, there is a large difference between the `basic agent`'s score and the other agent's scores. At these densities is where our algorithms are able to work out things that the `basic agent` is unable to. At other densities, the inferences are not successful as frequently.



Efficiency

What are some of the space or time constraints you run into in implementing this program? Are these problem specific constraints, or implementation specific constraints? In the case of implementation constraints, what could you improve on?

In the case of our `hyper_advanced agent`, we run into a large time constraint. This is because the `hyper_advanced agent` uses proof by contradiction when inference fails. This means that for each tile, the algorithm assumes that it is a bomb and tries to find a contradictory statement in the Knowledge Base. The algorithm then assumes the tiles is safe and again tries to find a contradiction. As a result of needing to do this for each tile, the algorithm is very time consuming. One way that we could greatly improve on the run time of the `hyper_advanced agent` is by only doing proof by contradiction if the KB has changed substantially since the last time we ran it. This way, we would not be unnecessarily rerunning the proof by contradiction and would be able to cut down on a lot of time.

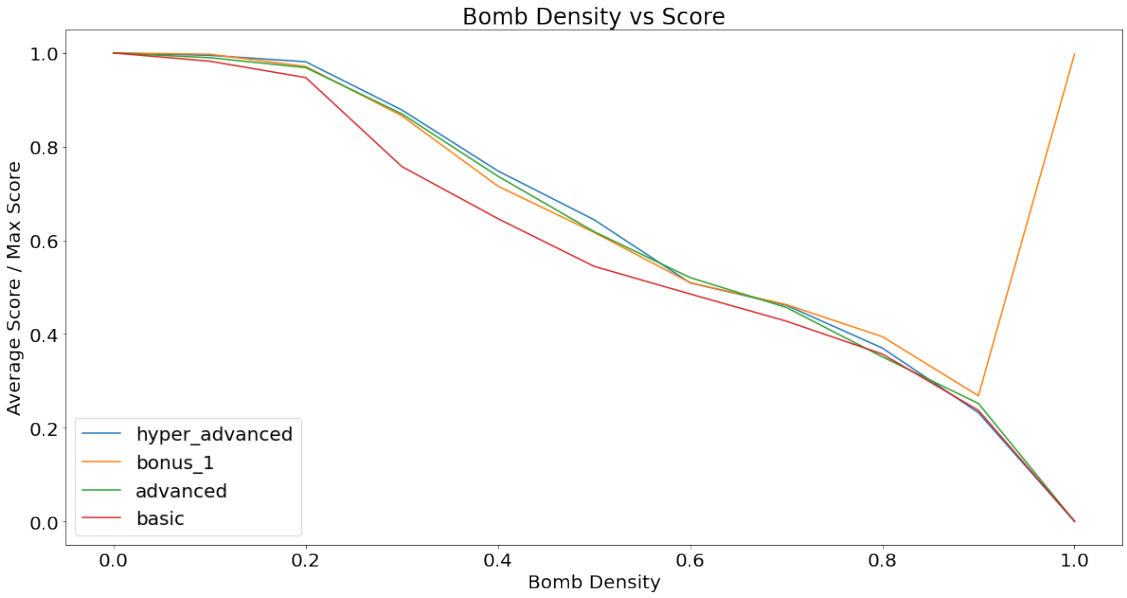
Bonuses

Global Information

Suppose you were told in advance how many mines are on the board. Include this in your knowledge base. How did you model this? Regenerate the plot of mine density vs average final score with this extra information, and analyze the results.

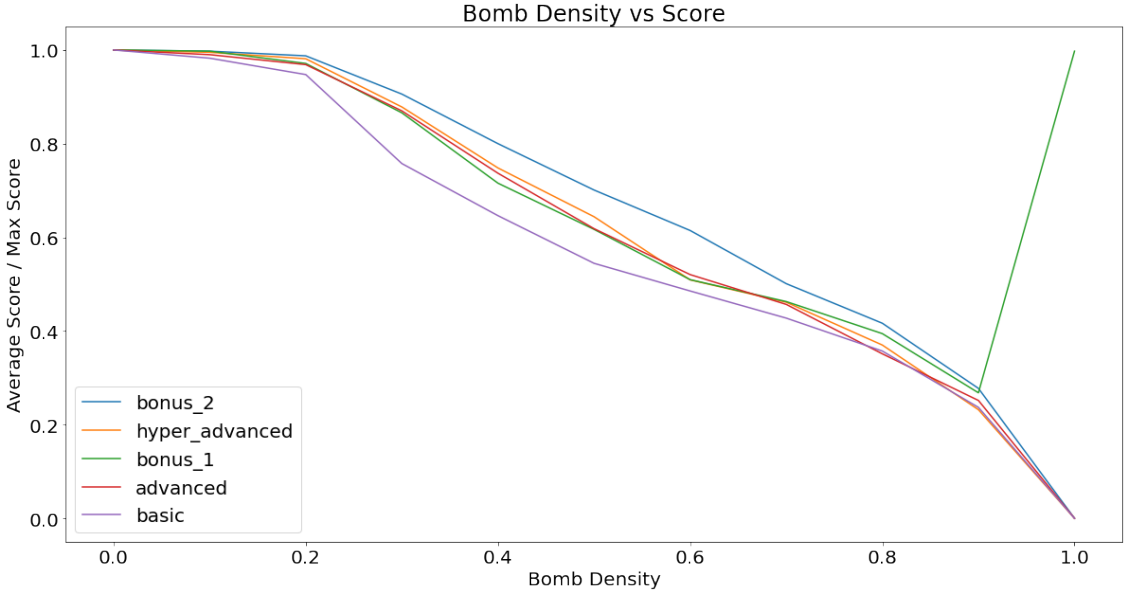
If the algorithm knew in advance how many mines are on the board, in the end game of the board, when the

rest of the tiles are bombs. When the number of unopened tiles is equal to the number of unaccounted for bombs, the algorithm automatically flags the remainder of the tiles and completes the board. We model this by adding another equation to the Knowledge Base that keeps track of how many bombs are not accounted for yet. In the graph below, we can see that the `bonus_1 agent` begins to perform better than the `hyper_advanced agent` and the `advanced agent` at around 0.8 bomb density. This is because at these high bomb densities, there is a high chance that in the end game of the board, the remainder of the unopened tiles are bombs.



Better Decisions

In both the basic and improved agent, when nothing more could be inferred, the agent selects a covered cell at random. Build a better selection mechanism. How can you justify it? Regenerate the plot of mine density vs average final score with improved cell selection, and analyze the results.



For a better selection mechanism, we implemented a new function `not_so_random_tiles`. This function is called instead of opening a random tile. The mechanism works by calculating a `prob_of_bomb` for each

unopened tile. If the tile is in an equation in the Knowledge Base, the `prob_of_bomb` for that tile is the value of that equation divided by the number of variables in that equation. If a tile is in multiple equations, we take the sum of the value of the equation it is in divided by the number of variables in that equation for all the equations that it is in. If the tile is not in any equations in the Knowledge Base, the `prob_of_bomb` for that tile is the number of mines divided by the number of unopened tiles. We then pick the tile with the lowest `prob_of_bomb` and choose that tile as the one to open. In the graph above, the `bonus_2 agent` performs significantly better than the `hyper_advanced agent` and the `advanced agent` from 0.3 to 0.9 bomb densities. This is because opening tiles while taking into account the probability that tile is a bomb greatly helps the agent to perform much better when no inferences are able to be made.

Clever Acronym

Our acronym for the advanced algorithm is Inferential Bomb Squad (IBS).

LaTeXed Report

This report was written and compiled using LaTeX.